DOCUMENT RESUME

ED 052 806 LI 002 951

AUTHOR Hart, Robert
TITLE Instant Fortran.
INSTITUTION Hofstra Univ., Hempstead, N.Y.
PUB DATE 11 Nov 70
NOTE 37p.; (1 Reference)
AVAILABLE FROM Author, New College of Hofstra, Hempstead, New York 11550 ($.50; $.40 each for 10 or more copies)

EDRS PRICE EDRS Price MF-$0.65 HC-$3.29
DESCRIPTORS Automation, *Computer Programs, *Computer Science Education, *Electronic Data Processing, *Information Processing, *Programing Languages
IDENTIFIERS *Fortran

ABSTRACT

As part of a short (five-hour) "package" or "module" of computer instruction, this booklet can be inserted into existing courses. It is aimed at the problem of giving large numbers of liberal-arts students a "literacy" in computation -- cheaply, feasibly, and with minimum facilities, staff, and administrative "blither." (Author/NH)

INSTANT FORTRAN

Robert Hart
New College of Hofstra University
Hempstead, New York 11550, U.S.A.
(November 11, 1970)

TO THE INSTRUCTOR: Classroom quantities of this booklet are available. Permission to reproduce or adapt this booklet will usually be freely granted for purposes which are educational, non-commercial, and not-for-profit. Written permission and the exact conditions must be secured in advance from the copyright holder.

A Teachers' Commentary on this booklet is (or will be) available from the author. This booklet is part of a short (five-hour) "package" or "module" of computer instruction, which can be inserted into existing courses; and which is aimed at the problem of giving large numbers of liberal-arts students a "literacy" in computation -- cheaply, feasibly, and with minimum facilities, staff, and administrative blither. Further information, including a "do-it-yourself kit" for instructors (of instructor's guide, student handouts, evaluation, bibliography, folksy advice, etc.), is available from the author.

ACKNOWLEDGEMENTS: Development of the package and this booklet was supported in part by the Undergraduate Science Curriculum Improvement Program of the National Science Foundation. The final section of this booklet owes a debt to Section 1 of G.E. Forsythe's excellent "Educational Implications of the Computer Revolution," in Applications of Digital Computers, edited by W. F. Freiberger and W. Prager (Ginn and Company, Boston, 1963).

ERRATUM: The bottom line of p. 5 should have a decimal point added and read

X = 2A - B + 1 , in Fortran this would be X = 2.0*A - B + 1.0 . (Note that the

The picture to keep firmly in mind when programming a computer, is of an idiot sitting at a desk calculator. The idiot is fast but very dumb. Your program is a series of instructions to the idiot, which the idiot obeys slavishly one after the other. If you could write a set of explicit instructions to a very dumb secretary for carrying out your calculation, that would be pretty nearly a computer program.

Almost any computer program has about the same structure: you tell the idiot to take in numbers, to do something with them, and to give you the results out. I will essentially show you one instruction for each of these, using the simplest program I can think of--a program to add two numbers together. But I will drop hints on how these innocent-looking instructions and the same structure can, in fact, be used to do socially important and sizable calculations.

Before we tell the idiot to take in two numbers, add them together, and give the result out, I will comment on something which often bugs people. We will be giving the idiot numbers, and a set of instructions for what to do with them. Both are necessary, but they are logically distinct. The idiot couldn't do much if we gave him instructions for adding two numbers together, but didn't tell him which two to add. He couldn't do much, either, if we gave him two numbers, but didn't tell him whether we wanted to add them or multiply them. The instructions (the program) sit on the desk in front of the idiot, and he follows them, one after the other. The numbers (data) are written one number to a card, on a stack of index cards, and this stack is somewhere else on his desk. Every once in a while he hits an instruction which tells him to do something with one of the numbers in the stack. However, these data are not themselves instructions. They are just there to be used like the instructions tell him to. The program is the recipe that the idiot follows, the data are the ingredients.

The point of this separation is that the same recipe can be used to do many jobs by putting in different ingredients. Once we have written a set of instructions for adding two numbers together, those instructions can be used to add together any two numbers, just by giving the idiot a different pair of data cards. (Also, since this leaves the instructions the same, their original translation into machine language can still be used. If the data were part of the program, changed data would mean a changed program, and computer time would have to be used to translate this new program.)

That isn't too interesting for adding two numbers, which no one in his right mind would do on a computer, anyway. However, if you have a complicated set of instructions--a payroll program, say--it means that the same instructions can be used to prepare another company's payroll. All that's needed is giving the idiot the different numbers (hours worked and hourly rates of pay, say) for the employees of the other company, and following the same set of instructions he'll grind out the payroll for the other company. (Some companies make a living this way, preparing payrolls for others. They are called computer service bureaus.)

Similarly, if the original company wants to prepare next week's payroll, they use the same instructions over. They don't have to rewrite the program, even if Jones got a nickel raise. They just slip the card with his old hourly rate out of the stack of data cards, and replace it with a card with his new rate written on it.

That is a Very Important Point about computers. For repetitive calculations, you get a large return of computation for a small amount of programming. If by hand or on a desk calculator you were to average two batches of numbers (of 1000 numbers each, say), getting the average of the second batch would be just as much work as getting the average of the first. On a computer, the instructions have to be written for averaging the first batch, but once this is done, later batches are

done essentially free, i.e., with little extra human effort. Indeed, computers are useful only for jobs that repeat in some way. If you want to do something one time only, it is easier to punch the keys of a desk calculator yourself, than to write the instructions telling the idiot how to do it.

Well, on to telling the idiot to take in two numbers, add them together, and give the results out.

Get Numbers In

We have given the idiot a stack of two data cards. Each card has written on it, in ordinary decimal notation, one of the numbers we want to add together. The first instruction to the idiot is

READ A

Program I

This tells the idiot to pick up the top card from the stack, read whatever number he finds there, memorize the number and call it A , and then throw this card into the wastebasket. Later on, whenever you want him to use A , he'll know what A is. Throwing the card into the wastebasket after he read it, means that the stack now has a new top card--the second card of the original stack.

To get in the second number, you do exactly the same thing. If you want to call the second number B , the second instruction to the idiot is

READ B

Program II

As before, this tells the idiot to read the top card from the stack, memorize whatever number he sees there and call it B , and to toss this card into the wastebasket. Note, tho, that he tossed the first card into the wastebasket when he obeyed READ A . So this top card is now the second card of the original stack.

So now the idiot knows what A and B are, and is ready to do something with them. His A and B are the numbers on the first and second data cards you

gave him. And that is about all there is to getting numbers in. At any point in the instructions that you want the idiot to read the next data card, you tell him READ W , where W is whatever you want to call that number.

Before going on to doing something with A and B , it may be useful to clean up some odds and ends. First, instead of A and B , you can use any letter (except I, J, K, L, M, or N) as the name of a number. They don't have to be in alphabetical order. (I, J, K, L, M, and N have other uses, which I hope to explain in a sequel to this booklet, "Son of Instant Fortran.")

You aren't restricted to taking in two numbers. You can take in as many as you want, and you can take them in anywhere in the program. For example, the four instructions READ A , READ B , READ S , READ Q , at whatever point encountered, would result in the idiot's there reading the next four data cards, and calling the numbers found on the cards A , B , S , and Q , respectively.

The data cards are punch cards (IBM cards), and not index cards as I used for illustration. However, that doesn't make much difference for understanding what is going on. The numbers are punched on them in a code of holes, in addition to being written on them. The computer reads the holes. The punching and writing is done by a keypunch, which is similar to a typewriter except that it punches in addition to writing. Learning to use it takes about five minutes.

Each data card has on it <u>only</u> a number, as, e.g., 62.7 , 19. , or 0.017 . It does <u>not</u> have on it A = 62.7 , B = 19. , or S = 0.017 . The computer knows which is which by <u>order</u>. It knows that A is 62.7 because this is the number on the top data card when the idiot reaches the instruction READ A . Putting in " A = " is unnecessary, illegal, and immoral. Instead of doing your job the computer will make insulting remarks.

Also, each of the data numbers must have a decimal point. Plain 19 without a decimal point is illegal. More about these later, but they are common enough errors to deserve double mention.

Do Something With Them

Most any program consists of getting numbers in, doing something with them, and getting the results out. We have gotten numbers in--the idiot knows the values of A and B --and now we will do something with them. This is the part of the program that does the work, but it is easy. The secret is that Fortran is designed to be as much like ordinary arithmetic as possible. To tell the idiot to add A and B and call the result X , we write X = A + B ; and similarly, all other calculations look like what they mean.

The program so far looks like

```
READ A
READ B
X = A + B
```

Program III

We told the idiot to take in A and B , and now that he knows what they are, the last instruction tells him to calculate their sum, and whatever number he gets, memorize that number and call it X . Later on, whenever you want him to use X , he'll know what X is.

Let us write down some other calculations that look like what they mean. Instead of X = A + B , we could have written X = A - B , and the idiot would have subtracted B from A and called the result X . X = A/B is division. (The slash is division.) X = A*B is multiplication. (The asterisk is used as the multiplication sign, to avoid confusion with the letter x.) These fundamental operations can also be used in combination. If, in ordinary algebra, you had X = 2A - B + 1 , in Fortran this would be X = 2.0*A - B + 1 . (Note that the

numbers must have a decimal point.) The idiot would do with it the same thing he did with X = A + B : calculate the thing on the right-hand side of the equal sign, using the known values of A and B , and whatever value that came out to, call it X . If, in ordinary algebra, you had $X = \frac{(14.3\ A - AB + 23B)}{(2A - 17)}$, in Fortran this would be X = (14.3*A - A*B = 23.0*B)/ (2.0*A - 17.0).
Note that all the multiplication signs (asterisks) have to appear explicitly; for example, in 14.3*A and A*B .

So I will continue writing X = A + B in the sample program I am developing, but what this really stands for is "as complicated a calculation as you want," written in the ordinary notation of arithmetic.

You can make the calculation more complicated in other ways. For example, you could add an instruction to the program to make it look like

```
READ A
READ B
X = A + B
V = X/A
```

Program IV

The point is that by the time the idiot hits the last instruction, he knows what X is, and he can use it to calculate further quantities, just like A and B . He happens to know X for a different reason than he knows A : X he calculated and A he read in from a data card. But once he knows a quantity, for whatever reason, he can use it to calculate further quantities. Remember too, that X = A + B and V = X/A could each be as complicated an expression in ordinary algebra as you want. Nor are you restricted to two letters on the right-hand side. Once the idiot knows A , B , and X , the last instruction could have been V = 2.0*B - A*X + 17.3 . The idiot handles all such instructions in the same way. You write, on the right-hand side, in what is essentially ordinary algebra, as complicated a

combination of previously-known quantities as you wish. The idiot calculates the whole mess on the right-hand side according to the notation of ordinary arithmetic, plugging in the previously-known values. Whatever number the mess comes out to, the idiot memorizes it and then calls it whatever letter you put on the left-hand side of the equal sign. Later on, whenever you use this letter, the idiot knows it has this value.

So I will continue using the uninteresting-looking X = A + B in the sample program I am developing, to represent the entire "Do Something With Them" part of the instructions; but this is a fake that stands for any of the more complicated calculations you can do using the same structure and ideas.

In fact, socially significant computer programs can sometimes be quite simple. The guts of a payroll program, for example, is PAY = TIME*RATE , no more complicated than X = A + B . (TIME is the number of hours worked, and RATE is the hourly rate of pay.) Programs like this have put people out of work (and created jobs for programmers). Not because they do anything that a third-grader couldn't, but because the computer does it quickly and repetitively, a point I will beat to death later.

PAY = TIME*RATE also illustrates that you can call numbers by names that look like what they mean, not only single letters like the A , B , and X that we have been using. (PAY is a single number just like A ; it is not a number P times a number A times a number Y .) This is an enormous practical help in writing real programs, where there may be many names of numbers. There are other uses. It is rumored that some programmers write their programs so as to embarass female employees and vice-presidents as much as possible. However, single letters are adequate for our purposes, so the details are in "Son of Instant Fortran."

PAY also shows why multiplication signs (asterisks) have to appear explicitly: in X = A*B , for example. If you wrote X = AB , the idiot would consider AB to be a single number called AB , just like PAY is a single number.

Before going on to getting the results out, a couple of odds and ends. If you write something like X = A/B + 1.0 , people often ask whether the idiot understands this as X = (A/B) + 1.0 , or as X = A/(B + 1.0) . The cheap answer, which always works, is to put in enough parentheses to make your meaning unambiguous. (Parentheses are on the keypunch, just like on a typewriter.) If you mean X = (A/B) + 1.0 , write that. If you mean X = A/(B + 1.0) , write that. More parentheses than necessary don't hurt, so use them freely whenever you are in doubt.

A slightly more expensive answer is that here, as everywhere, Fortran is as much like ordinary arithmetic as possible. So Fortran evaluates things like you did in grade school: First you do the multiplications and divisions, and then you do the additions and subtractions. So X = A/B + 1.0 means X = (A/B) + 1.0 .

Students sometimes ask what happens if you write READA or READ A , instead of READ A . Fortran sees these, and also perversions like RE ADA and R E A DA , all as the same thing. Spaces are irrelevant in the instructions (tho the numbers on data cards should be written in the ordinary way). They are useful because READ A is more legible to you than READA , X = A + B more legible than X=A+B , especially when you have a long series of such instructions; but the computer is equally happy with all of them. In other words, if you write everything (instructions and data numbers) in the form that looks natural to you, it will be correct. Fortran is deliberately designed to be as useful for people as possible.

Get Numbers Out

In most any program you get numbers in, do something with them, and get the results out. Continuing with Program III, we have gotten numbers in, and done something with them (" X = A + B "); and now we want to get X out. A human assistant might know that X is the desired result, and give it to you; but the computer is an idiot, and you have to tell it "give me this number." If you don't, the computer knows what X is, but it won't tell you.

The instruction for "give me X " is WRITE X . When the computer hits this instruction it prints the value of X on the printer. This is a kind of glorified typewriter attached to the computer, which prints out a whole line at a time. (With other instructions, or other computers, X might be printed instead on the regular typewriter that is part of the computer, or punched on to a punch card, or gotten out in other ways. Exactly which way isn't too important.)

With this addition, Program III becomes

```
READ A  } <- NUMBERS IN
READ B  }
X = A + B <- DO SOMETHING WITH THEM
WRITE X <- RESULT OUT
```

Program V

And that is about all there is to getting numbers out. At any point in the instructions that you want the computer to give you the value of a previously-known number, you say WRITE W , where W is the name of that number. You can get out as many previously-known numbers as you want, anywhere in the instructions, and they can be any numbers that the idiot knows (not just numbers he calculated). For example, you could have written

```
READ A
READ B
WRITE A
WRITE B
X = A + B
WRITE X
```

Program VI

Here the idiot would give you the values of A and B right after he took them in, before going on to calculate and print out X as before.

No paragraph

This has its uses. It is a check that the computer has read in the values of A and B that you intended.

Now we have taken numbers in, done something with them, and gotten the results out; and that is the basic structure of most real computer programs.

But that is still the hard way of adding two numbers together, or of doing any one-time calculation. If you want to do something one time only, it is usually easier to punch the keys of a desk calculator yourself, than to write the instructions telling the idiot how to do it. It becomes useful when you have lots of pairs of numbers to add together (or lots of employees on your payroll). If you have five hundred pair of numbers (or five hundred employees), it is not necessary to write Program V five hundred times. You write it once and tell the idiot to go back and do the same thing for the next pair of numbers (or the next employee), and the next, and the next, ..., and so on until the job is done. Thus you get a tremendous return of computation from a small amount of programming. That is the value of computers.

Contrary to popular opinion (and the opinion of most of your professors), speed is not their dominant merit; or at least speed would usually be useless without this repetitive ability. If the computer could only do one-time calculations, programming a calculation instead of doing it by desk calculator would be a waste of your time and of expensive computer time--no matter how fast the computer.

Before we go on to getting the program to repeat, some of the comments I made at the start of this booklet might now make more sense. These were about the difference between program and data, and about the usefulness of keeping them separate. The basic point was that once a program has been written to do a particular job, it can be used with little further human effort, to do that job on any numbers, just by giving the program different data numbers to chew on. (Another aspect of repetition.) If you look at Program V, you'll see exactly that. These instructions will add any two numbers together--just give the idiot two different data cards, and whatever numbers you put on them, the idiot will read in those numbers, add them together, and write out their sum. Which numbers you give him is totally unimportant. He'll do it for any numbers whatever.

Repetition

We have taken numbers in, done something with them, and gotten the results out. Now we want to tell the idiot to go back and do the same thing with the next pair of data numbers. We want him to go back to some preceding point in the program, so we give that point a name, and tell him GØ TØ that name. In our case, we want him to start over at the first instruction, so Program V becomes

```
37    READ A
      READ B
      X = A + B
      WRITE X
      GØ TØ 37
      END
```

Program VII

The names of instructions in Fortran are positive integers (no decimal point), smaller than 9999 . Other than that, 37 is a random number. We could have used 162 , 589 , 14 , or whatever. The main thing is that the two numbers have to match. We couldn't have said GØ TØ 85 if there were no instruction 85 in the program.

Also, END is at the end of every program, and I have put this in too. (Data numbers are not program; they happen to come after END .)

Incidentally, if we want the idiot to add together 500 pairs of numbers, of course we have to tell him what those numbers are; so we give him a stack of 1000 data cards instead of the stack of two data cards he had before, but that doesn't affect the program.

It may look odd that we can use the same instructions to add together five hundred pairs of numbers; the same letters A , B , and X in fact, each get used 500 times. The secret is that A , B , and X are not fixed names of numbers. If A gets the value 17.2 while doing the first pair, this does not mean that it has this value forever after. Rather A , B , and X are storage locations in the machine's memory--sort of boxes, or slots, or pigeonholes in the computer's

memory, in each of which one number can be stored -- and which can be erased and have a new number put into it. Thus, when the computer does the second pair of numbers, it erases the old numbers that were in the boxes A , B , and X as it goes along, and puts into them the values of A , B , and X for the second pair. It does this for each new pair of numbers, so it can go along forever adding pairs of numbers together, each time erasing the A , B , and X from the preceding pair.

This is the source of much of the computer's power. In a moment I will give a detailed example of what happens when Program VII is used to add together many pairs of numbers. But first I want to clean up some shorter items.

First, 37 is the <u>name</u> of an instruction. There is nothing numerical or sequential about it. SAM or JØE would make more sense as names of instructions, but Fortran insists that the names of instructions be integers like 37 . But there is nothing any more numerical or sequential about 37 than there would be about SAM or JØE . The next instruction after 37 does <u>not</u> have to be 38 . It could as well be 14 . In fact, you shouldn't give an instruction a name unless it needs one. The first instruction needs one because elsewhere we say GØ TØ that instruction, but so far no other instruction needs a name.

Similarly, if the next instruction that needed a name were the third instruction after 37 , it would <u>not</u> have to be 38 , and it would <u>not</u> have to be 40 . It could as well be 9674 .

A trivial point concerns the slashes in GØ TØ . This is to avoid confusion with the numeral zero. As far as the computer is concerned, O (zero) is a number like any other number, so if you give him GO TO (with zeroes), this doesn't make any more sense to him than G3 T3. The slashes avoid this confusion in handwriting; on the keypunch, just like on a typewriter, numeral zero is with the other numbers, and letter "oh" is with the other letters.

Everyone makes that mistake once, and it is particularly frustrating because the program <u>looks right</u>, dammit. Similar confusion occurs between handwritten numeral 1 and letter I ; and between numeral 2 and letter Z . Again, on the keypunch, numerals 1 and 2 are with the other numbers, and the letters are with the other letters. Handwritten these four are conventionally distinguished as 1 (plain vertical line), I (with top and bottom bars), 2 (the ordinary way), and Z (with a bar thru the middle, Continental style).

<u>Coffee Break</u>

Now we have come to a major break in this booklet. Program VII is a complete program which would actually run on some computers. Simple-minded tho it is, it was made that way on purpose, to show the principles easily. As advertised at the start, it takes numbers in, does something with them, and gets the results out; which is the structure of most real computer programs. It also displays the important <u>repetitive</u> ability of the computer. Despite its simple-mindedness, I have hinted how the same structure and ideas can do sizable and socially important calculations.

If you understand the logic of Program VII, you understand what is going on. Much of the rest of this booklet, except for the next section, is a cleaning-up of necessary details. Further -- unlike most everything up to now -- some of these details (and whether or not they are necessary) can differ on different computers. However, the differences are often small, and the logic behind them is the same in all computers' versions of Fortran. To illustrate the typical magnitude of these differences, some versions of Fortran use PRINT X instead of WRITE X , to do the same job; and some might use CALL READ (A) instead of our READ A , meaning exactly the same thing. But once you understand what is going on, that sort of change is easy.

Further, starting with the third section after this, the rest of this booklet deals with things that will be easier to understand when you actually do them: when you actually punch your program on to IBM cards and run it on the computer. So I will just give the minimum advance information you need, and leave most of the understanding to when it is easiest.

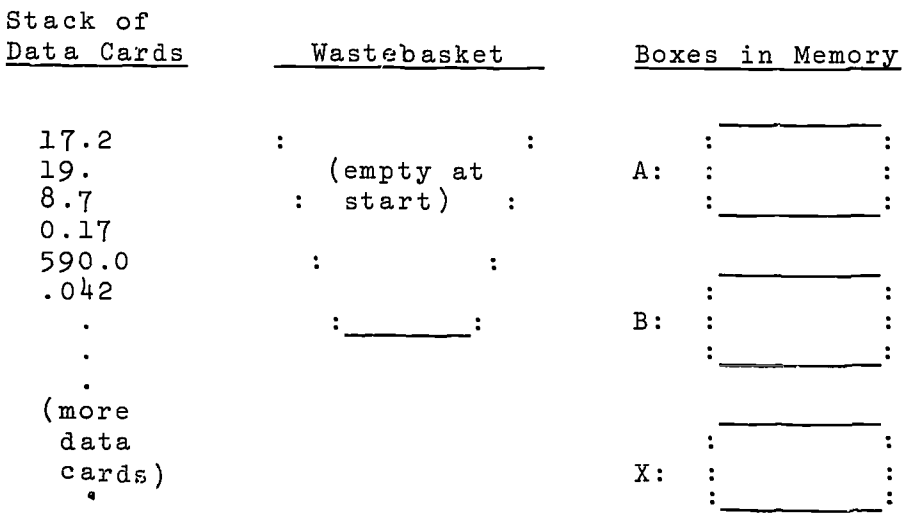
So the rest of this booklet will not be as inspiring, except perhaps for the final section on the social implications of computers. These details are necessary and reasonable, and some touch on things vital to serious computing; and I will try and show why this is all so. But gurnisht holfen, many of them are not as central to our purposes as what has gone before.

The next section is exempt from these maledictions. It gives a detailed example of how Program VII can add together many pairs

of numbers, not just one pair. If this seems clear from my earlier comments, you might skip it. A reasonable check is to ask yourself what numbers would come out if you gave Program VII the following stack of data numbers: 17.2, 19., 8.7, 0.17, 590.0, .042 . If you get 36.2, 8.87, and 590.042 (the sums of the first, second, and third pairs of data numbers), you may be all right. Otherwise, or if you are unsure how it got these, the step-by-step example of the next section should clear it up better than anything. Anyway, after you finish (or skip) the next section, is the time for a break.

Repetition II

On to the detailed example of what happens when Program VII is used to add together many pairs of numbers. The idiot has the program, and also a stack of data cards (one number to a card, with a decimal point in each), a wastebasket to throw the cards into after he has read them, erasable boxes in his memory called A , B , and X , and the printer on which he writes out results. They look like this at the start:



The idiot starts carrying out Program VII, and hits READ A the first time, which tells him . . . by now you should know what it tells him. The top data card with 17.2 written on it goes into the wastebasket, and 17.2 gets written in the box called A . Thus:

Data Cards	Wastebasket	Boxes in Memory	Printer
19. 8.7 0.17 590.0 .042 . . .	17.2	A: 17.2 B: X:	(still nothing)

The idiot then hits READ B the first time. The new top card with 19. on it goes into the wastebasket, and 19. gets written in the box called B :

Data Cards	Wastebasket	Boxes in Memory	Printer
8.7 0.17 590.0 .042 . . .	19. 17.2	A: /17.2/ B: /19./ X: / /	(nothing)

We have simplified the pictures of the wastebasket and boxes, to keep the secretaries from climbing up the walls.

Now the idiot hits X = A + B the first time, and this tells him to see what is in box A , see what is in box B , add these two numbers together, and whatever sum he gets, put it into the box in his memory called X :

Data Cards	Wastebasket	Boxes in Memory	Printer
8.7 0.17 590.0 .042 . . .	19. 17.2	A: /17.2/ B: /19. / X: /36.2/	(nothing)

Note that this is non-destructive: The idiot just looks at the numbers in boxes A and B ; he doesn't erase or alter them.

Now the idiot hits WRITE X the first time, and this tells him to go to the box in his memory called X , see what number is there (it is the sum of the first two data numbers), and write that number on the printer:

Data Cards	Wastebasket	Boxes in Memory	Printer
8.7 0.17 590.0 .042 . . .	19. 17.2	A: /17.2/ B: /19. / X: /36.2/	36.2

This is the first sign of life you get from the computer, i.e., the first of your answers printed out and given to you where you can read it. Note that this also is non-destructive: The idiot prints out whatever number he sees in box X , but doesn't erase or change the number in box X . So far everything is as before, because everything has been done the first time, with no repetition yet.

Now the idiot hits GØ TØ 37 the first time, starting the repetition. This sends him back to READ A to start over again from there. So he hits READ A the second time, and it means the same thing it did the first time: Pick up the top data card (now 8.7), throw it into the wastebasket, and put whatever number was on this card into box A , first erasing whatever number used to be there:

Data Cards	Wastebasket	Boxes in Memory	Printer
0.17 590.0 .042		A: /8.7 /	36.2
.	8.7	B: /19. /	
.	19.	X: /36.2/	
.	17.2		

The first erasing what used to be there is now important. It is the key to how the same A can be used to add together 500 pairs of numbers. Before, when the idiot hit READ A the first time, and box A was just blank, it could be ignored.

Now the idiot goes down the list of instructions, obeying one after the other, as before. He hits READ B the second time,

and this tells him the same thing: Pick up the top data card (now 0.17), throw it into the wastebasket, and put whatever number was on this card into box B , first erasing whatever number used to be in box B :

Data Cards	Wastebasket	Boxes in Memory	Printer
590.0		A: /8.7 /	36.2
.042	0.17		
.	8.7	B: /0.17/	
.	19.		
.	17.2	X: /36.2/	

Now boxes A and B hold the second pair of data numbers.

Note, tho, that the contents of box X is still the same. X = A + B is not an algebraic equality in the sense that the contents of box X always has to equal the sum of the contents of boxes A and B . It is an instruction which is only carried out when the idiot actually hits that instruction.

Now the idiot does hit X = A + B the second time and, as before, this instructs him to see what is in boxes A and B , add those two numbers together, and whatever sum he gets, put it into the box called X , first erasing whatever number used to be in X :

Data Cards	Wastebasket	Boxes in Memory	Printer
590.0		A: /8.7 /	36.2
.042	0.17		
.	8.7	B: /0.17/	
.	19.		
.	17.2	X: /8.87/	

Now box X does have in it the sum of the second pair of data numbers.

The situation with X = A + B is often summarized by saying that the equal sign in Fortran is replacement rather than ordinary algebraic equality. X = A + B instructs the idiot to calculate the right-hand side, and to replace the contents of box X with this number.

Having done READ A , READ B , and X = A + B the second time, the idiot keeps going down the list of instructions; next he hits WRITE X the second time, which tells him to see what number is in box X , and write that number on the printer:

Data Cards	Wastebasket	Boxes in Memory	Printer
590.0		A: /8.7 /	36.2
.042	0.17		8.87
.	8.7	B: /0.17/	
.	19.		
.	17.2	X: /8.87/	

Thus, the sum of the second pair of data numbers has been given to you on the printer.

And so it goes. The idiot next hits GØ TØ 37 the second time, which sends him back to do the whole thing over for the third pair of data numbers, and then the next pair, and the next, and the next . . . .

In fact, the perceptive student may have noticed that this program never ends -- which is not elegant, but it does the calculation. The program just goes on until it runs out of data cards. Some computers then start working on the next person's program,

usually first making insulting remarks; others come to a grinding halt and require personal attention to get them started again, which is grossly inefficient. In serious programming there are instructions to tell the computer that 500 pairs of data numbers are coming in; so when he has done it 500 times he should stop already, and know that this program is done. These are given in "Son of Instant Fortran."

Take your coffee break.

READ and WRITE

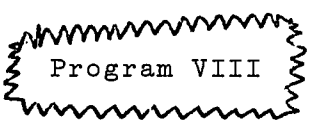
Program VII is a complete program, with all the logical structure needed, and would run on some computers. On most computers, however, including Hofstra's IBM 1130, two further pieces of information must be given the computer in the READ and WRITE instructions which get numbers in and out.

The first of these is easy to understand. Most computers have several ways to get a number out, and you have to tell the idiot which one. Each way is assigned a code number, so you say WRITE (3) X instead of WRITE X . The 3 tells the idiot to give you the value of X on the printer. If you wanted him instead to type X on the regular typewriter that is part of the computer, then instead of 3 you would use the code number for the typewriter; and similarly for other ways of getting numbers out.

Just as there are several ways of getting numbers out, so there are several ways of getting numbers in. So instead of READ A you say READ (2) A , where 2 happens to be the code number for the card reader that we will be using to take numbers in.

So Program VII becomes

```
37    READ (2,   ) A
      READ (2,   ) B
      X = A + B
      WRITE (3,   ) X
      GØ TØ 37
      END
```



The 2's tell the idiot to take numbers in from the card reader, and the 3 tells him to give you X on the printer.

The extra blank box in the READ and WRITE instructions is for the second piece of information that the computer needs when getting numbers in and out. This is harder to explain than the first piece. The basic point is where. When getting a number in, the computer has to know where on the data card to look for it. When getting a number out, the computer has to know where on the line to print it. This is important in serious programming, mostly because each data card or line of printed output has room for ten or twenty numbers, not just one. For example, you could get your results out in the form of a table. But where each number is, must thus be specified.

However, our interest is in doing the calculation, not in fancy ways of getting numbers in and out. So I will give you a simple all-purpose way that works for most ordinary decimal numbers, without explaining much about it: data numbers go anywhere in the seventh thru twenty-sixth spaces of the data card, one number to a card, with a decimal point in each number; results are printed out one number to a line. To do this, Program VIII becomes

```
37      READ (2, 14) A
        READ (2, 14) B
        X = A + B
        WRITE (3, 14) X
        GØ TØ 37
14      FØRMAT (6X, F20.9)
        END
        17.2
        19.
        8.7
        0.17
        590.0
        .042
```

program

data

Program IX

-

If your instructor told you to skip this section, you should mostly be looking at Program VII instead.

I have put in a stack of data cards after the program, for illustration.

Very briefly, the 14 in READ (2, 14) A tells the idiot that where for this READ instruction is specified by instruction 14 . Instruction 14 happens to specify that the first six spaces are skipped, and the number is in the next twenty spaces.

It is the same with READ (2, 14) B and WRITE (3, 14) X . In each case the 14 tells the idiot that where is specified by instruction 14 ; and instruction 14 specifies the same thing as above.

You may notice that instruction 14 is never actually obeyed by the idiot. Rather, instruction 14 gives the idiot information which he needs when he obeys the READ and WRITE instructions.

In line with giving you a simple all-purpose way that works for most decimal numbers, I have used the same where (specified by instruction 14) for getting all numbers in and out, but this is not necessary in general. I could add to the program other FØRMAT instructions with different numbers and specifying different

where's. Then READ (2, different number) B would tell the idiot, that where for B is specified by the FØRMAT instruction with that different number.

Incidentally, the 6X in FØRMAT (6X, F20.9) has nothing to do with the X in X = A + B , or in WRITE (3, 14) X . 6X is just the way to tell the idiot to skip the first six spaces.

Also, 14 is the name of an instruction, just like 37 was, and everything said about 37 is true of 14 . In particular, 14 is a random number, except that it has to be a positive integer smaller than 9999 . I could equally well have used 8934 thruout Program IX, instead of 14 .

Cards

You now have a program and data, like Program IX or Program VII,* written on a piece of paper. Computers don't read handwriting -- yet. They read IBM cards (and other things, but we will use IBM cards). You have to punch your program on to IBM cards, run the cards on the computer, correct any errors that turn up, and then try again. That is what the rest of this booklet is about. All this is more easily understood when you are actually doing it, so I will just give the minimum necessary advance information.

---

*Normally Program IX is your sample; but if Program VII or a near relative will run on your computer, your instructor will tell you to skip the preceding section on " READ and WRITE ," and stick with the easier Program VII: I will mostly talk about Program IX, but for you this means Program VII. If your instructor doesn't say anything, stick with Program IX.

Incidentally your first program should be nearly as simple as Program IX or VII. Misconceptions are easier to clear up that way. You might multiply or divide two numbers instead of adding them, but nothing much more elaborate.

The punching on to cards is done with the keypunch, which operates like a typewriter. You will be given a sheet of instructions when you actually do the punching. (Hang on to this religiously. It will enable you to use the keypunch on your own later, without further instruction). The keypunch punches your program on the cards in a code of holes, in addition to writing it.

Each line of Program IX or Program VII (including the lines of data) goes on a separate IBM card. (If you are using Program VII, it is followed by data after END , just like Program IX: Take a look at Program IX). These cards have 80 spaces. The name of an instruction (like 37) goes in the first thru fifth spaces. The rest of the instruction (like READ (2, 14) A) goes anywhere in the seventh thru seventy-second spaces. Data numbers come after END , and go anywhere in the seventh thru twenty-sixth spaces, one number to a card, with a <u>decimal point</u> in each number. (Remember that a decimal point is also required in numbers like the 1.0 and 2.0 of an instruction like X = 2.0*A - B + 1.0). This is the way Program IX has been spaced and decimal-pointed, so you can use it as an example. Also, the last page of this booklet is a full-size picture of a punch card, and of Program IX as it appears on cards.

These rules must be adhered to *E*X*A*C*T*L*Y*, as must all the commas, parentheses, and other "grammatical" details of Program IX. Otherwise interesting and instructive things will happen when you try and run your program on the computer.

Thus, Program IX has been transformed into a stack of punch cards, with 37 READ (2, 14) A on the top card, and .042 on the bottom card. This stack is pictured on the last page. Incidentally, the printing pictured on these cards (and also their color, and whether they have square or round corners), is immaterial to the computer. All it cares about is the positions of the holes. The printing is a convenience to you.

Extra Cards

Program IX has now been transformed into a stack of punch cards, shown on the last page. You will probably have to insert into it an additional card with your name and perhaps other identifying information on it, before the program is actually run on the computer; this prevents lost and anonymous programs. Details will be available.

Many computers, including Hofstra's IBM 1130, require that another sort of additional punch cards be inserted into the stack before the program is run on the computer. The main thing to know about these extra punch cards is that they are not worth worrying about. They are provided by the computer center, and are the same for all programs run on the same computer, at least at this beginning level. Thus, all you need is a sample program with the extra cards inserted, and you can use this as a sample for putting the extra

cards into any program. Initial samples will be available, and later your first program with the extra cards inserted will serve as a sample. Your first correct program is the other thing you should hang on to religiously if you may have further contact with the computer: It's the best possible sample for the extra cards, and for the spacing, decimal-pointing, and other grammatical rules of Fortran.

With the extra cards inserted, your program is ready to go into the computer. That is all you need to know about the extra cards, so you can skip the rest of this section. The extra cards do things like separate your program from the next person's program (obviously necessary if many programs are being run at once), and tell the computer that your program is a Fortran program (and not a machine language program, or a program in some other language). Thus these extra cards are not really part of your program, but rather instructions about your program; they have a kind of "traffic-directing" function in getting your program thru the computer. Since different computers have different internal "traffic patterns," the extra cards (unlike your Fortran program) can differ widely from one computer to another; but are identical for all simple programs run on the same computer, all of which follow the same route. Similarly, simple computers may get away without the extra cards, because all programs follow the same route, so you don't have to mark the route separately for each program.

Incidentally, these extra cards are ordinary punch cards, and you could punch them yourself on the keypunch if you wanted to;

but since they are the same for everybody the computer center duplicates them by the thousand and leaves stacks of them in appropriate places. More about these extra cards for the IBM 1130 in "Son of Instant Fortran."

Debugging

Your program has been transformed into a stack of punch cards with the extra cards inserted (if your computer needs the extra cards), this stack has been run on the computer, and -- surprise! -- your printed output from the computer is not your answers, but instead a lot of sarcastic remarks. That is all right. Contrary to popular opinion, finding and correcting errors ("debugging") is the major activity of programmers, not writing the program.

If your answers came out, keep reading anyway. They may not be right.

The computer tells you what it dislikes about your program by printing the name(s) (like 37) of the Fortran instruction(s) it finds objectionable, accompanied in each case by a code number for the type of error committed. This is straightforward when the offending instruction has a name, but not all do. If the computer dislikes an unnamed instruction, it goes back to the last preceding instruction with a name, and counts from there. Thus STATEMENT NUMBER 00037 + 002 means the second instruction after instruction 37 ; i.e., X = A + B in Program IX or Program VII. If there is no named instruction before the offending instruction, the computer counts from the beginning of the program: STATEMENT NUMBER 00000 + 004 means the fourth instruction from the start of the program.

Lists of the error code numbers, and the error to which each corresponds, will be available. Incidentally, the computer is usually correct in which instructions it dislikes, but is sometimes confused or not very helpful about why it dislikes them. So to find our why, it is often easier to first check the offending instruction against Program IX, to see what commas or parentheses you have misplaced, before consulting the list of error codes.

After you have found the errors you have committed, punch new cards and substitute them for the erroneous ones, and run the corrected program as before. You only have to repunch the cards that were in error, not the whole program. Repeat until the computer ceases to complain about your program.

Now you have a complaint-free program and, presumably, your printed computer output from this program is the number of answers you expected. Altho the printed computer output looks very impressive, and it apparently has the computer's seal of approval, these answers are not necessarily correct. The computer can catch grammatical errors (X = A ++ B , for example), but otherwise it does what you tell it to; and if you tell it to make mistakes, it makes mistakes, with great speed and obedience. If you meant to subtract two numbers and somehow wrote Program IX or Program VII to do this (i.e., you wrote X = A + B where you meant to write X = A - B), the computer won't complain about these programs, but the answers will be garbage.

There are many reasons why a complaint-free program may yield wrong answers, but fewer reasons why you should believe the garbage -- which is the real hazard. The most powerful single way

to keep from believing it, is hand-calculating a couple of answers and comparing them with the computer's answers. For example, trying Program IX or Program VII with the data numbers 1.0 and 1.0 , and getting the computer's answer 2.0 (instead of the correct answer 0.0 for subtraction), immediately catches the error that this is a program for subtraction. Do this with your program. After it gives you the right answers for test data, you can be moderately confident that it is giving you the right answers for other data too.

The above deals with errors in your program. Errors in your data are possible too (leaving out a decimal point, say). It is also possible to have a perfectly correct program with a perfectly correction instruction like X = A/B , which will work fine most of the time, but cause trouble for B = 0 .

It may be worth repeating that correcting errors is a normal part of programming. In fact, most programmers learn more from the computer's error messages ("sarcastic remarks") than they do from the programming texts. There is a serious moral to this -- one of the main morals of this booklet. Once you get over the hump of being able to get numbers in, do something with them, and get the results out -- which this booklet is supposed to teach you -- the computer will teach much of the rest. When in doubt about what a programming text is saying, ask the computer. Write a simple program that you understand completely, except that it makes one simple use of a feature of Fortran you are dubious about.

Even if you are wrong, YOU CAN'T HURT THE COMPUTER. You just get useful error messages. Because you can ask the computer, because each new feature is small, and because programming is logical, with a definite rule for (almost) everything, programming is peculiarly amenable to self-study; which is how I and many others learned it.

This section discussed what can go wrong with computer programs, which leads into

How to Make a Stupidity Amplifier.

The computer does what you tell it to, and if you tell it to make mistakes, it will make mistakes, with great speed and obedience. It is frequently said that a computer can in ten minutes do more calculations than an army of mathematicians working their whole lifetimes. It is less frequently pointed out, that in ten minutes it can make more mistakes than the army of mathematicians. A computer may be an intelligence amplifier, but it is also a stupidity amplifier.

Let us look at some of the intelligent and stupid uses of computers. Many of the stupid uses arise from regarding anything that comes out of the computer as divinely inspired, or at least too complicated for anyone except a computer programmer to argue with. Even if you are not an auto mechanic, you would doubt a story that a car jumped in the air, and did three loop-the-loops while singing "Hail Columbia." Similar nonsense circulates about computers, which are getting as important to our society as cars,

but few people have the same kind of general knowledge of computers that they have of cars. One aim of this booklet is to give a start toward that kind of knowledge.

As to the divine inspiration, you have seen that plain typographical errors can get in the way of this, like X = A + B instead of X = A - B ; and with more complicated programs the possibility of such mistakes, and their more subtle and troublesome relatives, increases greatly. However, here you know what you want to do, and only a flaw in the program makes that different from what you actually told the computer to do. Much sweat and checking should catch the flaw. Subtler questions are what you should do: not how to do it, but what is worth doing; and what value to place on the computer's answers when you have done it.

Data of doubtful accuracy or meaning are not purified by being handled on the computer. If you are uncertain of the values of A or B , or what they really mean (social scientists and psychologists take note), doing X = A + B on the computer instead of by hand, will not improve the accuracy or meaningfulness of X .

Similarly, a procedure of doubtful merit or relevance to your problem, does not gain anything by being done on the computer. If the sum of A and B has no bearing on your problem, it will still have no bearing even if you do X = A + B on the computer instead of by hand. These things are just as true of complicated calculations as they are of X = A + B -- no matter how impressive the computer output looks.

Two partial exceptions count toward the intelligent uses of computers. Using a computer can improve accuracy, in that once

you have a well-checked program, thereafter it will virtually always do that calculation correctly, unlike a human calculator (and do it essentially free: with little human effort). Also, if a procedure or data are basically relevant to your problem, and fail only because extracting that relevance is too lengthy for a human calculator, a computer may be able to do it. For example, treating more cases (more data) may make your result meaningful, and the computer's repetitive ability is peculiarly well suited to this. Similarly, a basically relevant procedure may require dubious approximations to make it short enough for a human calculator; the computer may be able to do it without approximations.

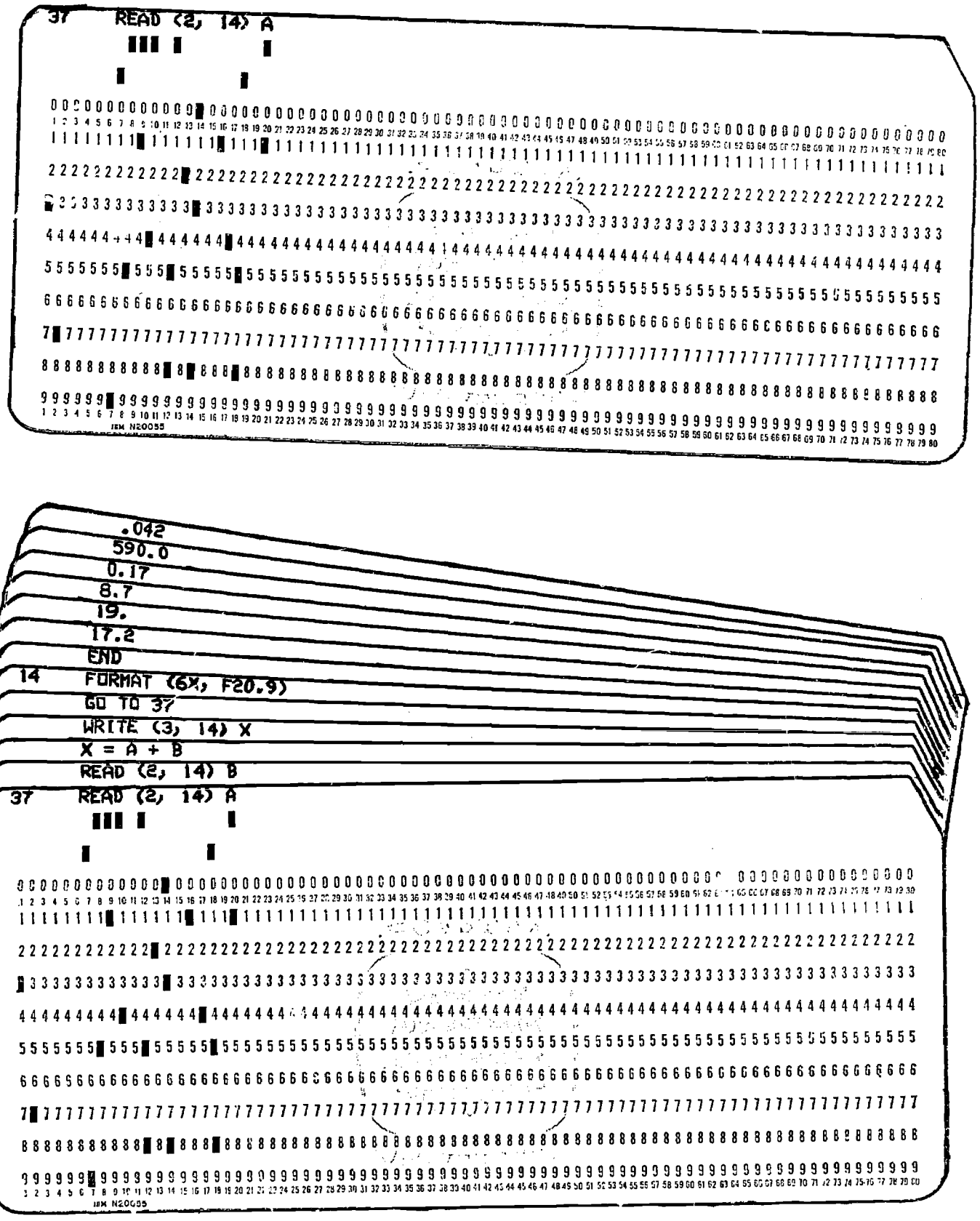
Again, however, the computer doesn't sanctify the lengthier calculation; it makes it possible. Whether it is worth doing is up to the user. Like all human tools, the idiot computer is only as good as the idiots who use it.

However, everything said so far about the uses of the computer is merely doing -- faster, oftener, easier, more accurately -- the same thing one would be doing otherwise. The really interesting uses of computers arise when its power makes possible new approaches to a problem -- new ways of thinking about it. For example, a car moves thirty times faster than a covered wagon. But the effect of cars on our society is not just that of a fast covered wagon. We do different kinds of things with cars, not just more of the same. For good and ill, cars have shaped our cities, our approaches to commerce, vacations, and the whole pattern

of our mobile society, in ways that an improved covered wagon could not. Cars have brought new industries, new kinds of jobs, certainly new kinds of problems and fears, new ways of thinking about the organization of society -- and their ultimate consequences are not understood yet.

Computers do the same thing as a dumb secretary -- ten million times faster, not thirty. For example, economic historians can now follow in detail 100,000 people or businesses instead of 100: an entire city or small state in detail. That changes your whole way of thinking about a problem. Computer-generated music gives composers any sound, and the audience will hear it as written, not filtered thru the abilities and interpretations of the performers. This latter may be as fundamental for music, as was the invention of writing for the art of storytelling. Computers would make possible daily voting on local and national issues, which would change our whole framework of thinking about government. A national data bank in Washington could contain the administrative, criminal, medical, and financial information about all citizens, which is now scattered in many records. It would be possible to monitor almost every business transaction in the United States. Checker-playing programs now learn to beat the people who programmed them, and computers are programmed to take intelligence tests and prove theorems in geometry: Their simulation of much of what commonly passes for thought, and even creativity, must sharpen our understanding of what human thinking is, and of what it means to be human.

The significance of computers is that fundamental possibilities like these are opening up in almost all fields. Again, exactly as with $X = A + B$ and its lengthier relatives, the computer doesn't sanctify these applications; it makes them possible. Whether they should be done, and how to use the computer intelligently and humanely in exploring them, is up to the user. Like all tools back to the caveman's axe, the computer amplifies man's possibilities -- for stupidity or intelligence.


37 READ (2, 14) A
IBM N20055
.042
590.0
0.17
8.7
19.
17.2
END
14 FORMAT (6X, F20.9)
GO TO 37
WRITE (3, 14) X
X = A + B
READ (2, 14) B
37 READ (2, 14) A
IBM N20055